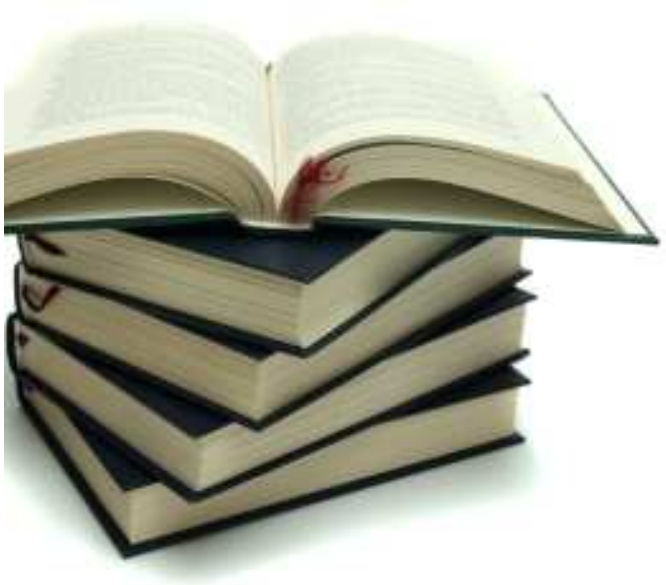
so2
kernel crunch

Cursul 3



3

Apeluri de sistem

12 martie 2009

Test Curs 2

- Care este entitatea VFS care identifică în mod unic, în sistem, un fişier?
- Ce problemă încearcă să rezolve planificatorul de operații de I/E?
- De ce în Windows există un API unic pentru operații asyncrone şi de semnalizare, indiferent de tipul resursei cu care se lucrează (timer, socket, fişier, proces, etc.), iar în Linux nu ?

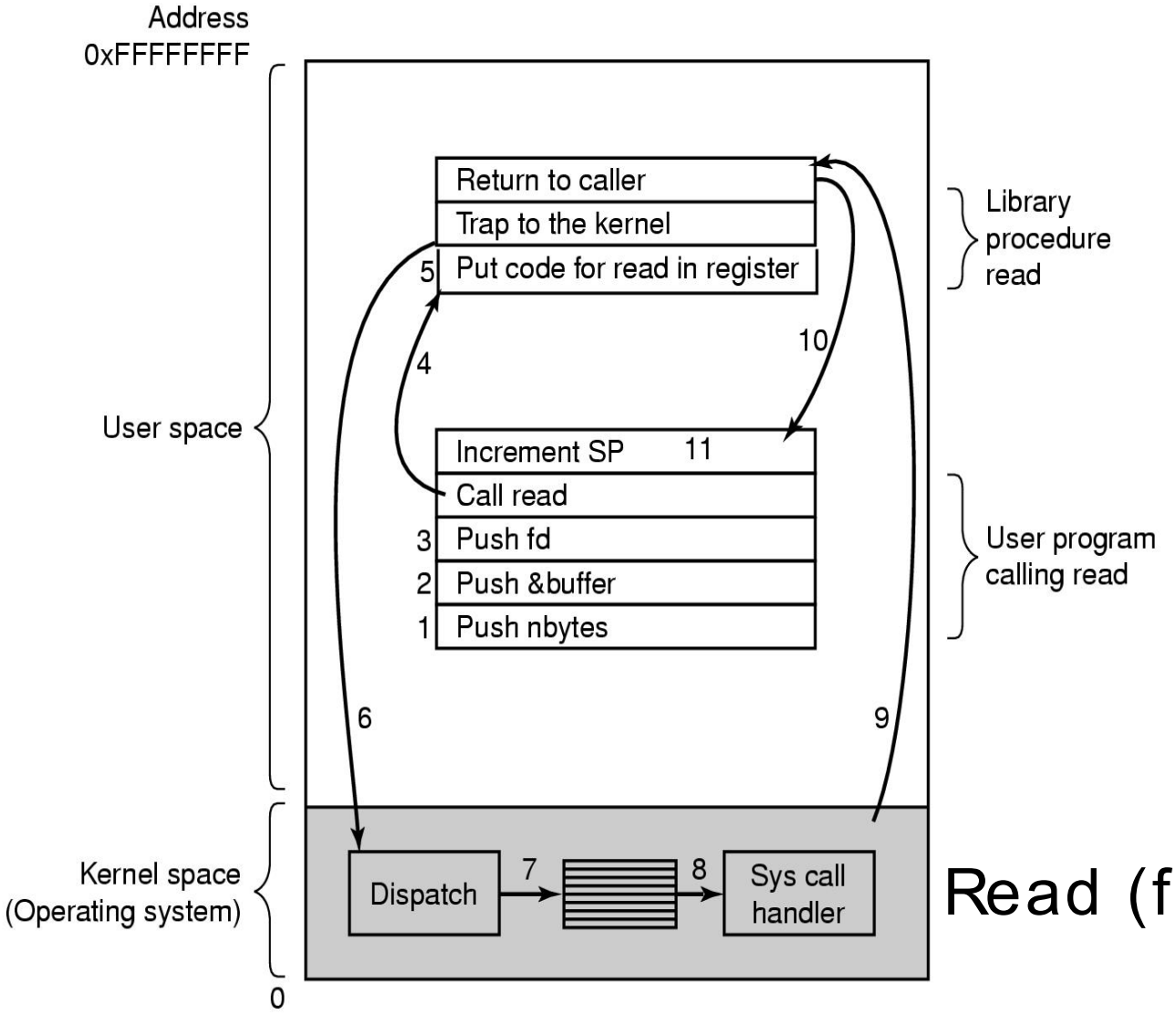
Apeluri de sistem

- Apeluri de sistem
- GAS, stack frame
- Implementarea apelurilor de sistem în Linux

Bibliografie

- UTLK: capitolul 10
- LKD: capitolul 5
- WI: capitolul 3 (System Service Dispatching)

Apeluri de sistem



Apelurile de sistem sunt modalitatea prin care aplicațiile accesează serviciile puse la dispoziție de kernel.

Read (fd, buffer, nbytes)

Exemple

- Exemple de funcții ce fac apeluri de sistem:
  - Linux: read, write, etc.
  - Windows: NtReadFile, NtWriteFile, etc.
- Exemple de funcții ce nu fac apeluri de sistem:
  - Memcpy, strcpy

Ce este un apel de sistem?

- O modalitatea de a intra în mod controlat în kernel: trap / excepție / întrerupere sincronă cauzată de execuție unei instrucțiuni speciale
- Trap-ul are asociat o rutină de tratare (sytem call dispatcher)
- Apelurile de sistem se identifică prin numere
- Informații suplimentare se pot pasa prin parametri

Ce se întâmplă la un apel de sistem?

Un apel de sistem determină comutarea contextului de la aplicație la kernel. Procesul curent îşi continuă execuția în kernel mode şi rulează rutina asociată apelului de sistem.

Ce se întâmplă la un apel de sistem? (2)

- Se trece din user-mode în kernel mode
- Se ridică restricțiile de accesarea kernel space
- Se schimbă stiva user cu cea kernel
- Se verifică şi copiază argumentele din user space în kernel space
- Se identifică şi rulează rutina asociată cu apelul de sistem
- Se comută înapoi în user mode şi se continuă execuția aplicației

Parametrii apelurilor de sistem

- Întregi de dimensiunea cuvântului maşinii – pentru simplitate
- Sunt stocați fie în regiştri, fie pe stiva user; sunt copiați pe stiva kernel de către dispatcher
- Pot conține pointeri (in/out); accesarea pointerilor se face de către rutina ce implementează apelul de sistem
- Valorile parametrilor nu pot fi considerate ca fiind valide – trebuie verificate;
- Atenție sporită la pointeri

Pointeri către kernel-space

- Aplicația poate pasa un pointer care să pointeze în kernel-space
  - Pointer de tip in către kernel space: acces la date protejate
  - Pointer de timp out: coruperea memoriei kernel
- Soluție: verificare valorii pointerului şi interzicerea apelului de sistem în cazul în care acesta pointează către kernel space

Pointeri către zone invalide

- Aplicația poate să paseze parametri cu pointeri către zone de memorie nemapate în spațiul de adresă al procesului
- Soluția evidentă: verificarea adresei înainte de orice accesare
  - Costisitoare, necesită căutarea adresei în spațiul de adresă al procesului

Soluția eficientă

- Nu facem verificări explicite
- Dacă o adresă este invalidă în momentul accesării sale se va genera un page-fault
  - Se rulează rutina de tratare a page fault-ului
  - Identificăm faptul că page falt-ul a fost cauzat de o adresă invalidă primită din user-space
  - Întrerupem apelul de sistem cu eroare
- În cazul în care adresa este corectă nu trebuie să face verificări suplimentare

Page fault-uri

- Copy on write, demand paging, swap
  - Identificare: adresa de fault este în user-space şi este validă
- Kernel bug
  - Identificare: adresa de fault este în user-space şi este invalidă
- Pointer invalid primit din user-space
  - Identificare: adresa de fault este în user-space şi este invalidă
- Ambiguitate în cazul ultimelor două situații

Exemplu ambiguitate

- Kernel bug

```
int *i=NULL;
if (*i !=  0)...
```

- Pointer invalid pasat din user-space

```
/* ubuf vine din user-space si e NULL */
memcpy(kbuf, ubuf, len);
```

Rezolvarea ambiguității

- Secvențele de cod ce accesează direct spațiul utilizator sunt marcate special
- La rezolvarea page fault-ului verifică dacă adresa codului ce a generat page-fault-ului face parte dintr-o zonă special marcată
  - Da -> pointer invalid pasat din user-space
  - Nu -> kernel bug

Analiză

- Soluția ineficientă:
  - Cost adresă validă: căutare adresei în spațiul de adresă al procesului
  - Cost adresă invalidă:căutare adresei în spațiul de adresă al procesului
- Soluția eficientă:
  - Cost adresă validă: zero
  - Cost adresă invalidă:cost page fault + căutare adresei în spațiul de adresă al procesului + căutarea adresei ce a generat fault-ul

Accesul la spațiul utilizator

- Nu se face doar în rutinele ce implementează apelurile de sistem ci şi în ... device drivere
- Nu accesați niciodată direct spațiul utilizator – folosiți funcțiile / infrastructura pusă la dispoziție de către kernel

64/32bit user/kernel

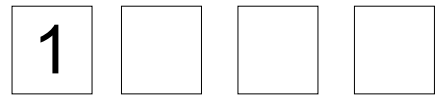
OS	char	short	int	long	void*
Unix 32bit	1 byte	2 bytes	4 bytes	4 bytes	4 bytes
Windows 32bit	1 byte	2 bytes	4 bytes	4 bytes	4 bytes
Unix 64bit	1 byte	2 bytes	4 bytes	8 bytes	8 bytes
Windows 64bit	1 byte	2 bytes	4 bytes	4 bytes	8 bytes

- Datele îşi pot schimba semnificația între user space şi kernel space

Exemplu

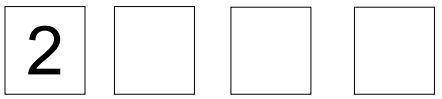
```
struct my_struct {
    long a;
    int b;
};
```

Little endian



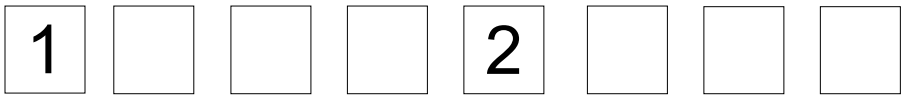
User-space 32 biți:

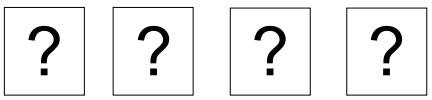
```
{ .a = 1, .b = 2 };
```



Kernel space 64 biți:

```
{a = 8589934593,
 b = -1075374248}
```





Apeluri de sistem în Linux

- Fluxul de execuție: aplicație -> libc -> apel de sistem
- Generarea trap-ului: `int $0x80` sau `sysenter` (Pentium II)
  - Numărul apelului de sistem: eax
  - Parametrii: ebx, ecx, edx, esi, edi, ebp
  - Rezultatul: eax

Exemplu de invocare

NAME

dup2 - duplicate a file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Exemplu de invocare (2)

```
000c7590 <__dup2>:
   c7590:       mov    %ebx,%edx
   c7592:       mov    0x8(%esp),%ecx
   c7596:       mov    0x4(%esp),%ebx
   c759a:       mov    $0x3f,%eax
   c759f:       int    $0x80
   c75a1:       mov    %edx,%ebx
```

Sintaxa gas (AT&T)

- Comentarii
  - Pe mai multe linii: /* */
  - Pana la sfarsitul linei (dependente de platforma): # ! ; |
- Operanzi:
  - Imediati: precedati de $
    - Intel: push 4
    - AT&T: push $4
  - Registri: precedati de %
    - Intel: push eax
    - AT&T: push %eax

Sintaxa gas (AT&T) (2)

- Operanzi (2)
  - Sursa si destinatia sunt inversate:
    - Intel: add eax, 4
    - AT&T: add $4, %eax
  - Dimensiunea operandului este specificata prin postfixarea b, w, l, q la mnemonica instructiunii:
    - Intel: mov al, byte ptr FOO
    - AT&T: movb FOO, %al

Sintaxa gas (AT&T) (3)

- Adresare indirectă
  - Intel: [base + index*scale + displ]
  - AT&T displ(base, index, scale)
- Exemple
  - Intel: mov eax, [100]
  - AT&T: mov 100(,1), %eax
  - Intel: mov eax, [ebx-100]
  - AT&T: mov -100(%ebx), %eax
  - Intel: mov eax, [100+ebx*4]
  - AT&T: mov 100(,%ebx,4), %eax

Frame stack

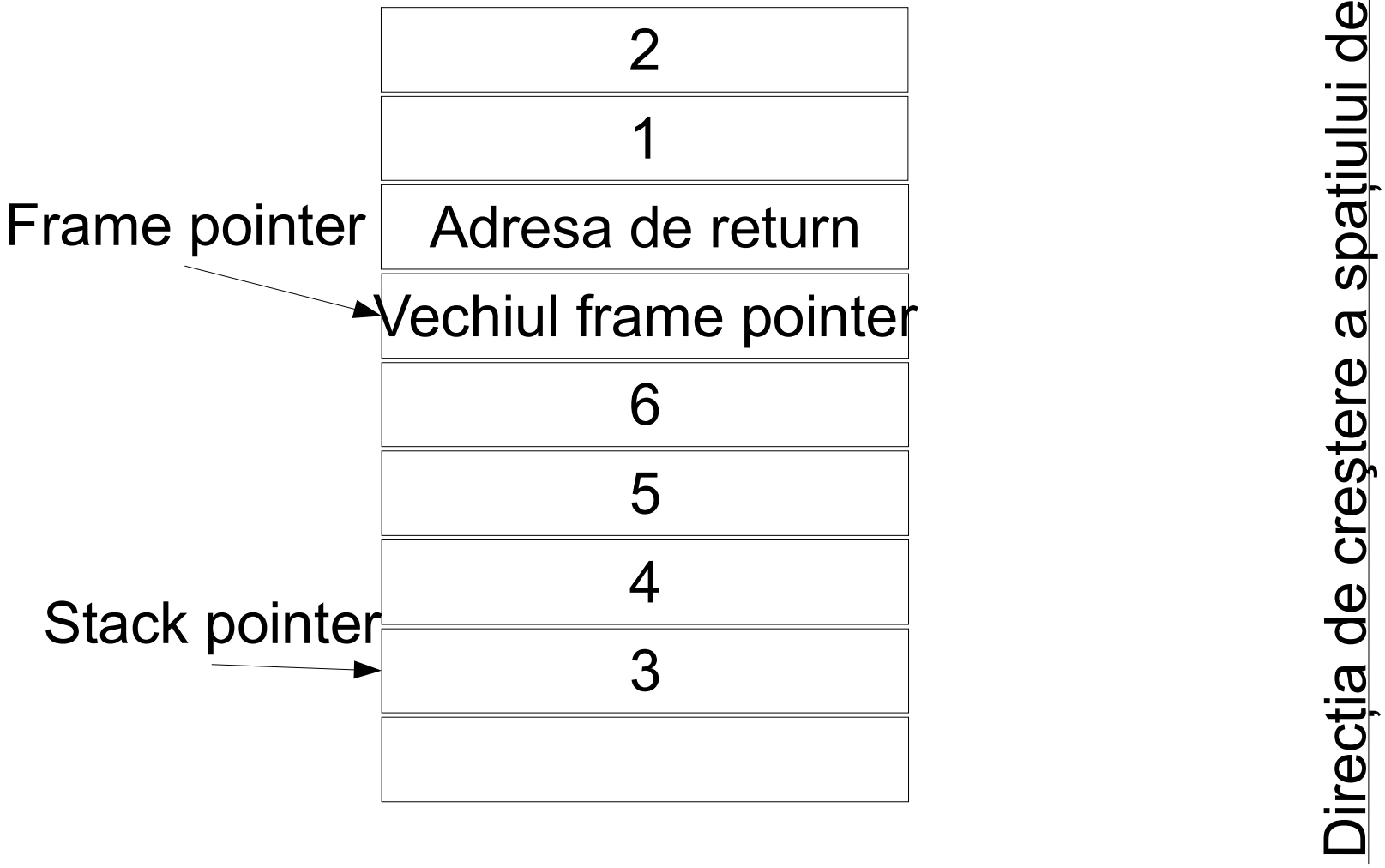
- Modul în care sunt aranjate pe stivă informațiile pentru funcția curentă
- Stiva programului este o stivă de frame-uri
- Frame stack-ul conține:
  - Parametrii funcției
  - Adresa de return
  - Opțional: adresa vechiului frame pointer
  - Variabilele locale

Exemplu

```
int a(int b, int c)
{
    int d=3, e=4, f=5, g=6;
    return b+c+d+e+f+g;
}

int main()
{
  return a(1,2);
}
```

Stack frame-ul pentru funcția a



Stack frame-ul pentru funcția a (2)

```
(gdb) break 4
Breakpoint 1 at 0x8048366: file f.c, line 4.
(gdb) run
Starting program: /home/tavi/a.out
Breakpoint 1, a (b=1, c=2) at f.c:4
4               return b+c+d+e+f+g;
(gdb) x/8 $esp
0xbfecae94: 0x00000003 0x00000004 0x00000005 0x00000006
0xbfecaea4: 0xbfecaeb8 0x0804839f 0x00000001 0x00000002
```

Invocarea unei funcții

```
$objdump -dr a.o
....
  44:    sub     $0x8,%esp
  47:    movl    $0x2,0x4(%esp)
  4f:    movl    $0x1,(%esp)
  56:    call    57 <main+0x21>
          57: R_386_PC32  a
...
```

Accesul la parametri şi variabile locale

```
00000000 <a>:
   0:   push   %ebp
   1:   mov    %esp,%ebp
   3:   sub    $0x10,%esp
   6:   movl   $0x3,-0x10(%ebp)
   d:   movl   $0x4,-0xc(%ebp)
  14:   movl   $0x5,-0x8(%ebp)
  1b:   movl   $0x6,-0x4(%ebp)
  22:   mov    0xc(%ebp),%eax
  25:   add    0x8(%ebp),%eax
  28:   add    -0x10(%ebp),%eax
  2b:   add    -0xc(%ebp),%eax
  2e:   add    -0x8(%ebp),%eax
  31:   add    -0x4(%ebp),%eax
  34:   leave
  35:   ret
```

Exemplu de invocare (2)

```
000c7590 <__dup2>:
   c7590:       mov    %ebx,%edx
   c7592:       mov    0x8(%esp),%ecx
   c7596:       mov    0x4(%esp),%ebx
   c759a:       mov    $0x3f,%eax
   c759f:       int    $0x80
   c75a1:       mov    %edx,%ebx
```

Rutina de dispatch

```
ENTRY(system_call)
        ...
        SAVE_ALL
        ...
        cmpl $(nr_syscalls),%eax
        jae badsys
        call *sys_call_table(,%eax,4)
        movl %eax,PT_EAX(%esp)
...

ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
```

Rutina de dispatch (2)

```
#define SAVE_ALL \
        ... 
        push %eax \
        push %ebp \
        push %edi \
        push %esi \
        push %edx \
        push %ecx \
        push %ebx
```

Implementare apel de sistem

```
#define asmlinkage CPP_ASMLINKAGE
  __attribute__((regparm(0)))

asmlinkage long sys_open(const char *
  filename, int flags, int mode)
```

- Forțează toți parametrii pe stivă (altfel compilatorul poate face optimizări)
- Funcția poate fi definită cu mai puțin de şase parametri (chiar dacă dispatcher-ul salvează cei şase regiştri pe stivă de fiecare dată)

Exercițiu

Scrieți secvența de cod ce interceptează apelul de sistem open şi face astfel încât la fiecare invocare să se afişeze numărul şi rezultatul apelului de sistem.

Rezolvare

```
struct syscall_params {
   long ebx, ecx, edx, esi, edi, ebp, eax;
};

asmlinkage long interceptor(struct syscall_params sp)
{
        int syscall=sp.eax, r=f(sp);
        printk(„%d: %d\n”, syscall, r);
        return r;
}

f=sys_call_table[NR_open];
sys_call_table[NR_open]=interceptor;
```

Rezolvare (2)

- Este doar un exercițiu didactic, în practică nu se recomandă interceptarea apelurilor de sistem
  - Probleme: race-uri
    - http://www.watson.org/~robert/2007woot/
  - Folosiți infrastructura de trace sau LSM
- Cazuri speciale pentru care această abordare nu funcționează
  - clone() - se uită şi la alți regiştri salvați
  - exec() – trebuie să modifice valoarea EIP-ului salvat de procesor la comutarea contextului

Intrebări

?